OrderFlow Technical Architecture

version 3.6.2

Realtime Despatch Software Limited

August 02, 2016

Contents

OrderFlow Technical Architecture

1. Introduction

OrderFlow is an enterprise-strength order processing and warehouse management system (WMS) with a wealth of configuration options and implementation possibilities.

This document provides details of the architecture of the system at a technical level. It presents a number of different architectural views to depict different aspects of the system.

1.1. Audience

The intended audience of this document is:

- business leaders who are considering OrderFlow as a solution to the needs of their organisation
- potential administrators of the OrderFlow system
- designers and developers of the OrderFlow system
- those responsible for supporting and deploying OrderFlow

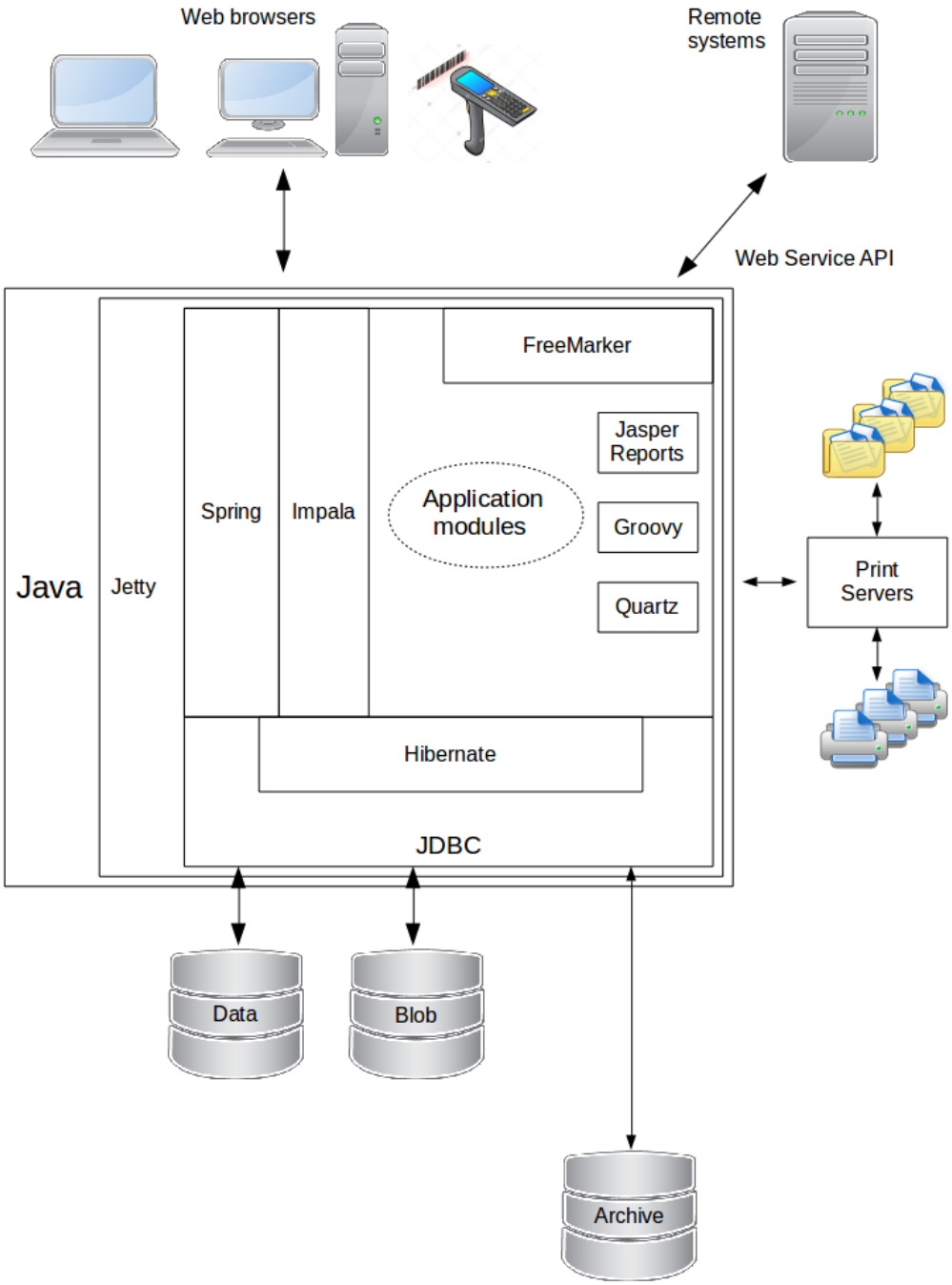
2. Overview

OrderFlow is a modular, layered, multi-interface application. It is written mostly in Java, and runs against a MariaDB relational database. It exposes desktop, mobile and handheld user interfaces (via HTTP or HTTPS), accessible from a web browser. It also exposes an XML over HTTP application programming interface (API).

The user interface framework follows the model-view-controller architectural pattern, supported by the Spring application framework, which is modularised using the Impala dynamic modular productivity framework.

OrderFlow is typically deployed within its own standalone Jetty web server, but can also be deployed to run within a separate web server (e.g. Apache Tomcat). It is almost exclusively deployed in Linux-based environments, most commonly using Debian or Ubuntu distributions.

The following diagram shows the technical architecture of OrderFlow at a high level.



The following notes explain various aspects of this diagram:

- Some of OrderFlow's logic is externalised into scripts written in the Groovy programming language.
- Most of the *views* (i.e. presentation logic) are written using the Freemarker templating language, to produce HTML, XML, JSON or other data formats.
- Scheduling in OrderFlow is controlled by the Quartz Scheduler Java library.
- OrderFlow facilitates some printing operations via a secondary **Print Server** Java application, also developed by Realtime Despatch. This communicates with OrderFlow's internal API to download print jobs and data files to machines local to the warehouse. Print files are then routed to the appropriate network or local printer, data files exchanged with external systems within the warehouse.
- Remote systems that OrderFlow communicates with include shopping cart systems, courier systems, enterprise resource planning (ERP) systems and accounting systems. Such communication can be as a client or server, from OrderFlow's perspective.
- Data extracted from the database can be presented in reports, which can be in textual data formats such as comma-separated values (CSV), spreadsheet format (XLS), or more sophisticated documents using the Jasper Reports Java library.
- Access to the MariaDB database is via Java Database Connectivity (JDBC) and the Object/Relational Mapping framework Hibernate ORM.
- When persisting data, OrderFlow typically separates 'file' data into a separate database, called a Blob (Binary Large Object) database. This is to keep the main (i.e. non-file) database from growing too large. Data from the main database can be *archived* to another database or the file system, if required.
- OrderFlow can also be configured to run in a clustered configuration for high availability.

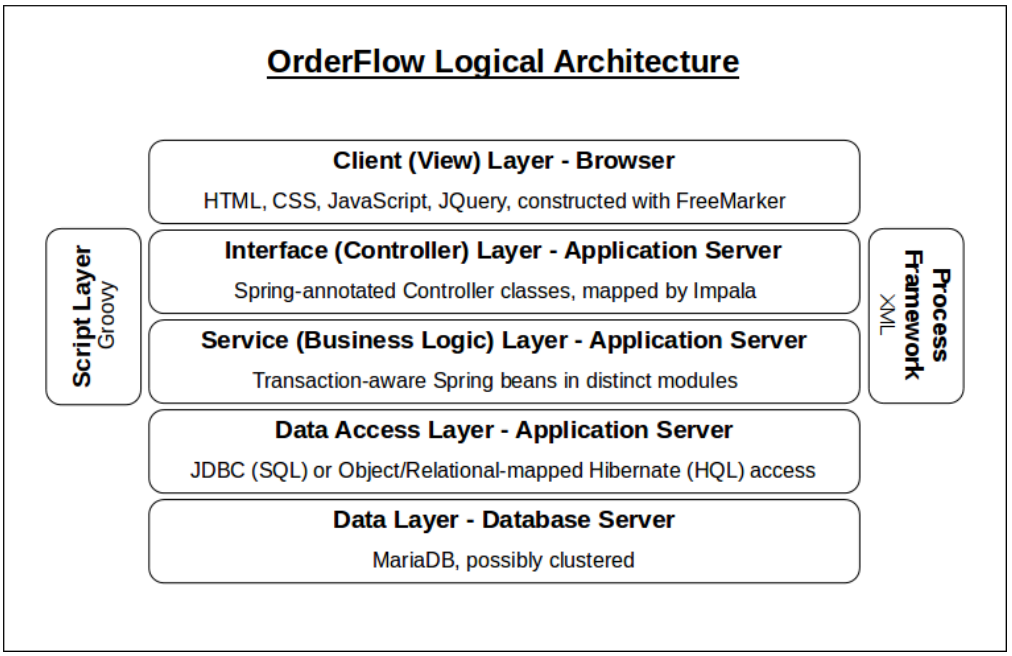
The next sections of this document will present different architectural views of OrderFlow.

3. Logical View

This section provides a logical view to OrderFlow's architecture.

OrderFlow is divided into logical layers, each having a particular responsibility. This approach isolates various system responsibilities from one another, therefore simplifying both system development and maintenance.

The following diagram shows the logical layers of the OrderFlow system, from the client layer down to the database layer. An explanation of the layers follows.



3.1. Client Layer

The **Client Layer** of OrderFlow renders the user interfaces that are presented to users in their own browsers. There are three main user interfaces:

- **Desktop** (also commonly called the *web* interface)
- **Handheld**
- **Mobile**

Each user interface is presented to the browser as HTML code, which the browser renders. The HTML makes use of Cascading Style Sheets (CSS) for styling, and JavaScript combined with JQuery for the more complex interactional logic. OrderFlow creates the HTML code (and responses in other formats) using the FreeMarker templating language. This allows dynamic content to be inserted into the resulting HTML page.

The *Desktop* interface is designed to be presented in a PC-based web browser, with standard and wide versions detected from the browser window size.

The *Handheld* interface is designed to be presented in a smaller browser on a rugged mobile computer, e.g.:

The *Mobile* interface is designed to be presented on a browser on a smart phone.

Interaction with the server is via HTTPS, using either GET or POST methods for page retrievals and form submissions respectively. In-page interaction is achieved using AJAX requests, which typically handle response data in the JSON format.

OrderFlow makes use of HTTPS *sessions*, to maintain user-specific data throughout the period in which a user accesses the application (without having to keep logging-in). It can also deposit *cookies* on the user's browser, to persist user preferences across different sessions. One example of this is to set the user's *workstation*.

3.2. Interface Layer

OrderFlow's **Interface Layer** has primary responsibility for serving the client layer with the data to render the user view. This typically includes dynamic data that is associated with the user's session, and also dynamic data to do with what that the user has requested to view (e.g. details of a shipment).

The interface layer uses the *Service Layer* and the *Data Access Layer* to source this dynamic data.

OrderFlow follows the *Model-View-Controller* architectural pattern, supported by the Spring application framework. It contains many 'Controller' classes, annotated with Spring's *Controller* stereotype. Collaborators are auto-wired in using the *Resource* annotation, and methods serving accessible URLs are annotated with the *RequestMapping* web binding annotation. ('Collaborators' are other objects that a Controller class may need to source data from, or to perform other functions - such as objects in the service or data access layers.)

Each method that serves an accessible URL can declare common parameters, which the Spring framework automatically populates. Examples of such parameters are:

- **Model** (Map) - the context passed between the Client and Interface layers, containing all dynamic data. Both the client and the server can modify this context.
- **HttpServletRequest** - an object representing the HTTP request made by the client browser.
- **HttpSession** - an object to hold user-specific data across more than one HTTP request.

The mapped request methods would typically modify the model for interpretation by the client layer.

The code for the interface layer is typically found in modules with the following prefixes:

- rtd2-web
- rtd2-view
- rtd2-remote

There may also be interface layer code in *bespoke* modules, which provide functionality specific to individual customers of Realtime Despatch.

3.3. Service Layer

The **Service Layer** of OrderFlow contains all the business logic and functionality that is not categorised as interface or data access functionality.

It is defined by the Java interfaces present in the **rtd2-api** module, some of which are described here:

- *ApiRequestGenerator* - generates requests to submit to remote APIs
- *BatchManager* - manages the creation, workflow and processing of shipment batches.
- *CourierManager* - provides all courier-specific functionality
- *EventManager* - provides functionality for submitting events in OrderFlow
- *DataImporter* - defines how data is imported into OrderFlow
- *OrderStockAllocator* - defines interface for allocating stock to order lines
- *DespatchNoteGenerator* - generates despatch notes for shipments
- *PropertyManager* - manages access to and update of application properties
- *ReportManager* - high-level interface for generating reports
- *ScheduleManager* - manages the scheduler and the invocation of scheduled jobs
- *CrudService* - provides basic create/read/update/delete service
- *StateTransitionManager* - manages state for transactional entities
- *UserManager* - manages all user logins and logouts
- *InventoryManager* - responsible for providing the inventory picture
- *StockMoveTaskManager* - provides methods for managing stock move tasks

The service layer also defines the **domain objects** in use within OrderFlow. These domain objects are mapped onto database tables by the data access layer. There are well over 200 such domain objects, so listing them here would be quite verbose. However, it is worth describing the categorisation of domain entities:

- **Transactional entities** such as **shipments**, **deliveries**, **import batches** and **print items**. These are business-relevant entities which are constantly being created during normal day-to-day operation of the system.
- **Incidental entities** such as **schedule executions**, **user sessions** and **log entries**. These are not business-relevant but are constantly being created.
- **Reference data entities** such as **locations**, **products**, **properties** and **report configurations**. These are relevant to a business but typically do not change as often as transactional entities.

These entity classes are all defined in the **rtd.domain** package within the **rtd2-api** module.

3.4. Data Access Layer

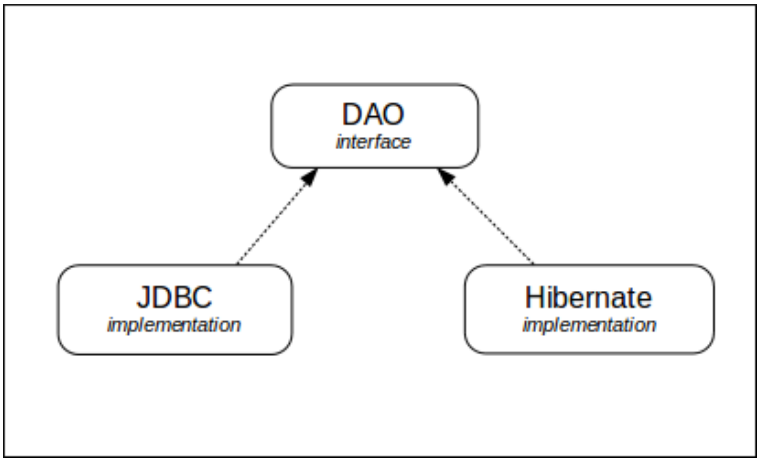
The **Data Access Layer** is the layer of code with responsibility to provide the higher logical layers with a mechanism to create, read, update and delete (CRUD) data in the data layer.

These responsibilities are shared across a single **CrudService** interface (defined in the **rtd2-service** package), and many **Data Access Object** (DAO) interfaces defined in the **rtd.dao** package.

Typically, each data access object interface is largely responsible for managing access for a single entity. The exception to this is the generic **DAO** interface, which acts on *all* entities.

The DAO interfaces are mainly concerned with *reading* the data in a convenient way for the other application layers to use. The *CrudService* and the generic *DAO* interface are mainly used for the creation, deletion and update operations.

The DAO interfaces are implemented either 'directly' via Java Database Connectivity, or through the Hibernate object/relational mapping framework. The JDBC implementations use SQL to query the data layer, whereas the Hibernate implementations use the Hibernate Query Language (HQL) to do this. The following diagram depicts this structure.



The Hibernate implementation can provide some automatic **filtering** when reading data objects from the data layer. This filtering ensures that only objects that the user has privilege to see are returned to the service or interface layer. This privilege is applicable at the organisation, channel and site levels, or any combination thereof.

Both implementations automatically restrict the **number** of data objects returned from the data layer. This is to maintain system performance by avoiding the potentially crippling effect of holding tens of thousands of objects in memory.

3.5. Data Layer

The **Data Layer** of OrderFlow consists of one or more relational databases. The number varies according to the configuration and size of the system, which is driven by the order volumes that the system is required to support.

The preferred relational database engine used by OrderFlow is MariaDB, an open-source, drop-in replacement for MySQL, which historically has been the most widely deployed open source database system. Although, MySQL is also still used in some OrderFlow deployments, MariaDB is now favoured mainly due to its superior performance in certain areas.

This list details the potential databases in use by OrderFlow (as mentioned in the *Overview* section):

- **Main database** – this typically contains all of the data used by OrderFlow, including transactional and incidental entities, and reference data. For a simple, low-volume instance, this also includes file data. For a higher-volume instance, the file data will be stored in a separate *Blob* (Binary Large Object) database. OrderFlow can be configured to store files in the main database or the blob database.
- **Blob database** – this is used to store file data, e.g. generated reports, API message request and response contents, imported file contents etc. The benefit of using a separate database is that the main database is not impacted by having to store and retrieve relatively large data records, while trying to serve users and other clients.
- **Archive database** – OrderFlow can be configured to archive data from the main database into an *archive* database. This contains a subset of the tables in the main database, as not all data is eligible (i.e. useful) for archiving. If this database is not in use then OrderFlow will archive data from the main database to the file system, or it will simply delete it.

As one would expect from a relational database, the integrity of the data in the main database is maintained through many foreign key relationships. Indexes on relevant database table columns ensure that data retrieval remains performant.

3.6. Script Layer

The **Script Layer** of OrderFlow allows various logical aspects of the system to be defined by run-time Groovy scripts, rather than pre-compiled Java code. Such scripts can be used in *application properties*, or they can be applied to various configuration objects, such as import handlers or location selection definitions.

The benefit of this feature is that OrderFlow can be controlled in a very fine-grained manner Additionally, such control can be changed at run-time, without the need for a new version of the application to be built, and even without the need for the application to be re-started.

3.7. Process Framework

OrderFlow's **Process Framework** provides another mechanism to change the logical behaviour of the system without recourse to rebuild or restart it. It is more powerful than the script layer in that entire processes can be configured at run-time. These processes may query reports, use the results to submit API requests, update entities, invoke operations and so on.

The framework uses XML documents to define processes, ensuring that they are human-readable. Such XML documents can be tested (within OrderFlow) before being applied to live entities.

More details on this can be found in the **Process Framework Guide**.

4. Process View

This section provides a process view of OrderFlow's architecture.

Processing is invoked by one of the following events:

- User activity on one of the user interfaces.
- Client activity on one of the application programming interfaces (APIs).
- Internal schedule firing.

The user/client activity will result in HTTPS request/response interaction, which, using Java Servlet Sessions, can handle multiple users and requests simultaneously. The service layer of OrderFlow is designed to be thread-safe, so each user request will only have access to data (in memory) for that request.

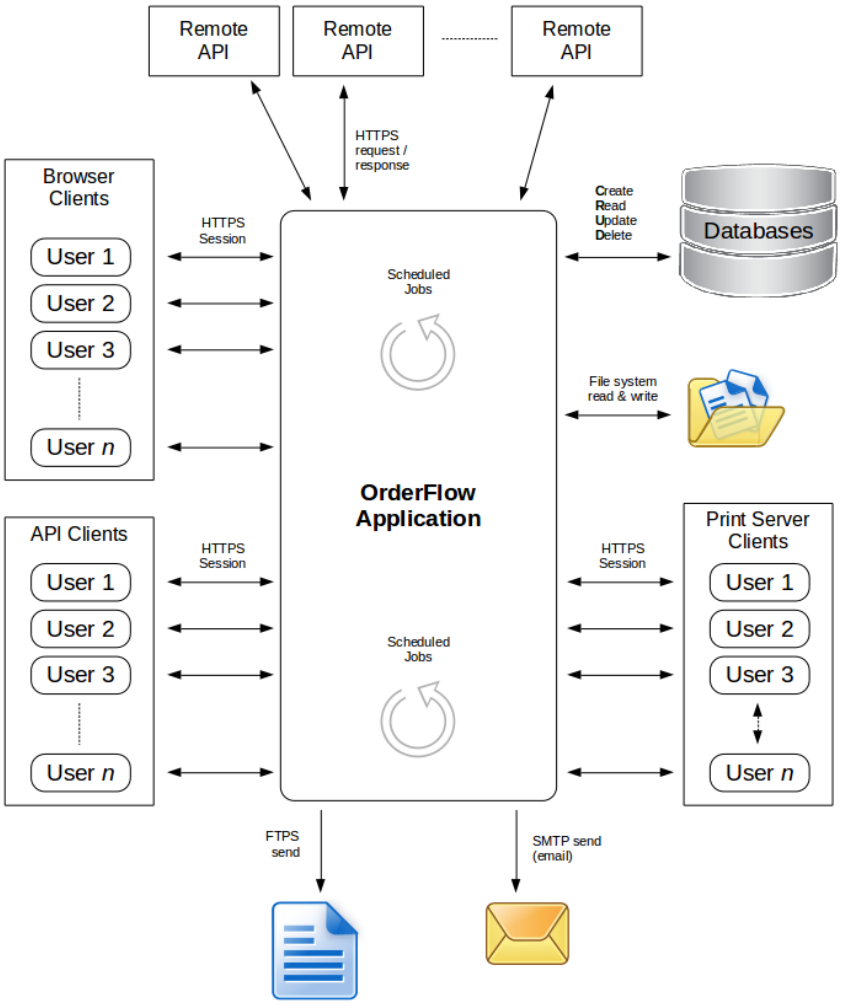
Additionally, the underlying database technology (via a database connection pool) allows concurrent, thread-safe data access.

Processing may result in one or more of the following events:

- Database CRUD operation.
- API request on remote web service.
- File system read or write.
- File sent to FTP server.
- Email sent to SMTP server.
- Response to original request, if applicable.

The following diagram shows the process flows in OrderFlow.

OrderFlow Processes



5. Development View

This section provides a *development* view of OrderFlow's architecture.

OrderFlow is a **modular** application, whose modularity is delivered using the Impala dynamic modular productivity framework (now hosted on GitHub). This allows the separation of source code into manageable parts that share a similar function.

The modular approach to the way the system is constructed also serves an important commercial purpose in that it allows bespoke functionality to be built in a way that does not affect the rest of the code base. This allows OrderFlow to be adapted to quite bespoke customer needs without introducing any unnecessary complexity or inconsistencies in the way the rest of the system works.

Modules are loaded when the application is started, and only functionality that is present in loaded modules is available to execute.

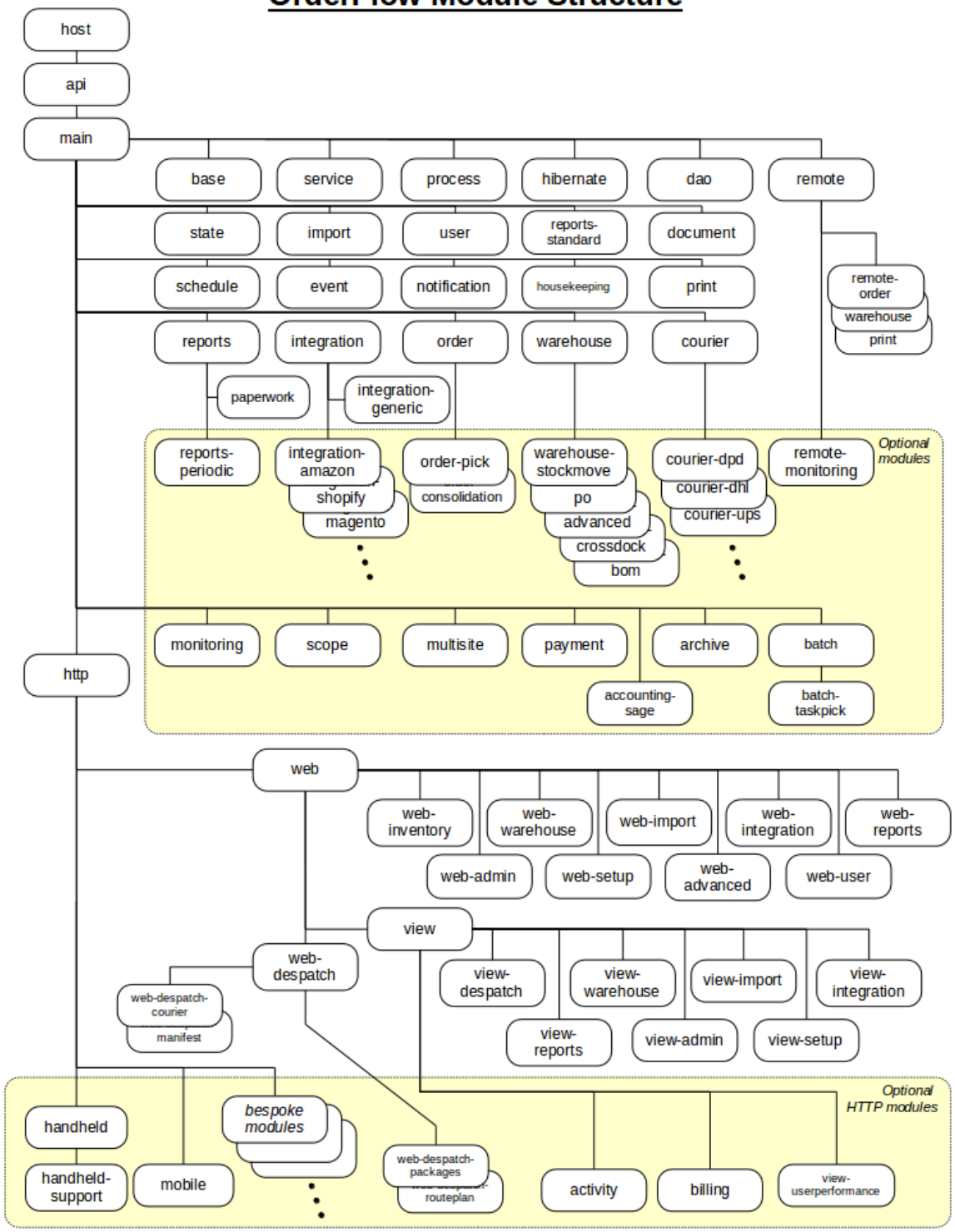
OrderFlow has the concept of *core* and *optional* modules. *Core* modules are included within the base distribution of OrderFlow, so are available automatically to all OrderFlow customers. *Optional* modules contain more advanced features that are made available by commercial agreement according to the needs of the Realtime Despatch customers.

5.1. Module Structure

OrderFlow's modules are organised in a hierarchical structure, with child modules inheriting functionality from their parents.

The following diagram shows the module structure of OrderFlow; an explanation follows the diagram.

OrderFlow Module Structure



5.1.1. Top-level Modules

Host Module

The top-most module in OrderFlow is the **host** module. This is what the web server container runs, since it contains the application's deployment descriptor (i.e. its *web.xml* configuration file). The *host* module also contains other configuration such as the *Spring* application context, plus other system, data access and logging properties. (These properties can be overridden in a deployed system by entries in files on the filesystem.)

The *host* module also contains the following:

- The application's landing page, i.e. *index.htm*.
- Source code for environment and performance logging, servlet filters, session termination and data source loading.
- The OrderFlow printing applet Java Archive (JAR) file.
- All JavaScript, Cascading Style Sheets (CSS), built-in images/icons and the application's URL icon.
- Version information for the application.

API Module

The **api** module of OrderFlow contains no Spring bean implementations, but instead defines most of the **interfaces**, **entity classes** and other **non-entity classes** used for encapsulating multiple data objects into a single object.

All other modules (apart from *host*) have visibility of the interfaces in the *api* module. Implementations of these interfaces may be present in any of the underlying modules.

Main Module

The **main** module contains more interface declarations, plus utility classes that are useful for many of OrderFlow's modules. It also contains a handful of *api* interface implementations, including universally-useful implementations of templating and report configuration interfaces.

The *main* module imports a significant number of Spring bean implementations (e.g. DAO implementations) into its Spring context file, in order to make them visible to all sub-modules. This prevents all sub-modules from having to repeat the same import statements.

5.1.2. Core Modules

The **core** modules in OrderFlow comprise of modules that are essential to the running of the application. They provide the basic, minimum functionality that the system offers.

Modules such as *order*, *warehouse*, *courier*, *reports*, *notification* and *service* contain core functionality, without which orders would not be progressible through the system.

Additionally, the DAO implementations in the logical data access layer are contained in the *dao* core module.

5.1.3. Optional Modules

Optional (or 'non-core') modules in OrderFlow contain functionality that is not essential for the running of the system, but enhances it in some way.

For example, different couriers and integration solutions are supported in their own modules. Also, OrderFlow's multi-site, multi-channel/organisational capabilities are present in the *multisite* and *scope* modules respectively.

If these optional modules are not loaded in a particular instance of OrderFlow, the functionality will not be available. This optional nature allows the deployed application size to be reduced where necessary, and it also

provides a greater degree of control over where optional functionality is deployed. This can help Realtime Despatch to enforce commercial arrangements, such as giving exclusivity over certain functionality.

5.1.4. HTTP Modules

The **http** module and its sub-modules are all concerned with supporting OrderFlow's user interfaces. The core http modules support the *desktop* interface - those prefixed with 'web' are loosely divided into modules according to the top-level tabs in the desktop interface.

Modules prefixed with 'view' are concerned with providing the **dashboard** functionality throughout the desktop interface. This functionality allows what appears on OrderFlow's user interface to be tailored according to a specific customer's needs, without needing to rebuild or even restart the application.

5.1.5. Optional HTTP Modules

Similarly to the optional *core* modules, the optional *http* modules contain user interface functionality that is not essential for the running of OrderFlow, but add additional features.

These additional features include the **handheld** interface, which is a user interface dedicated to run on ruggedised handheld warehouse computers. Not all customers use handheld devices in the warehouse, so this module is considered optional.

Finally, modules that contain *bespoke* functionality for specific customers are optional, for obvious reasons. These modules can contain functionality that may or may not be geared towards the user interface, i.e. they may also contain tailored internal logic if a particular customer requires this. Separating bespoke functionality into dedicated customer modules defines clear boundaries and provides a layer of insulation from the core system functionality.

6. Physical View

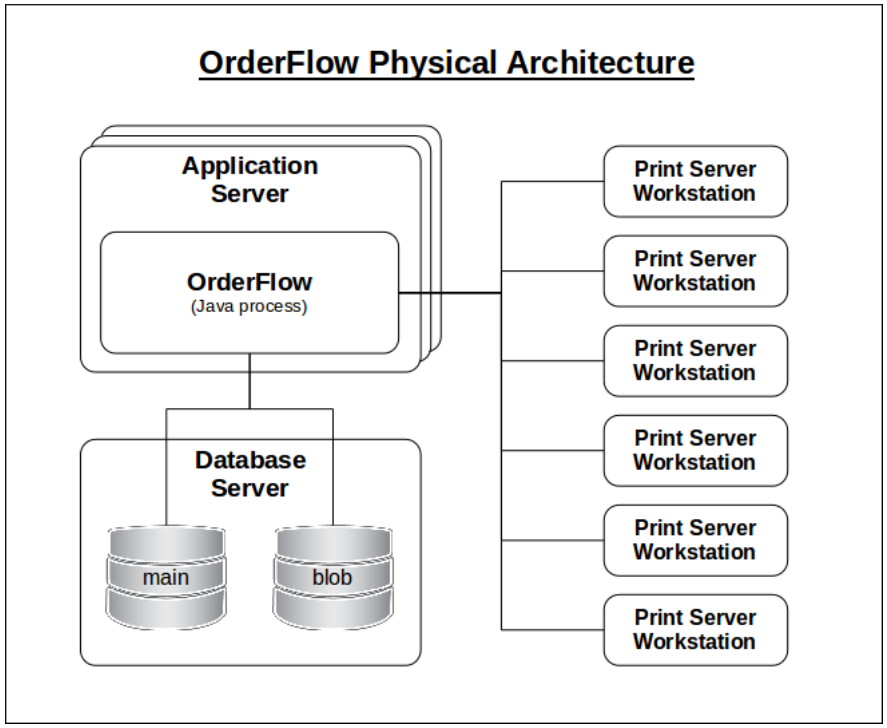
This section provides a *physical* view of OrderFlow's architecture.

OrderFlow is a Java web application that is (usually) deployed as a standalone Jetty web server, on an **application server**. This application server can be a physical server or a virtual machine (VM), typically hosted by a dedicated hosting company.

It runs atop a MariaDB database, which may or may not reside on a separate machine to the application server. If it *does* reside on a separate machine (physical or virtual), this is known as the **database server**.

Although not strictly within OrderFlow's boundaries, but known to interact with OrderFlow, are Print Server application instances, which reside on separate physical client workstations, typically Windows PCs in a customer's warehouse.

The following diagram depicts how OrderFlow is deployed at a *physical* level. Note that the multiple application servers are intended to depict the fact that OrderFlow can be configured to run in a clustered configuration.



7. Data View

As detailed in the *Data Layer* section, OrderFlow consists of one or more relational databases, running in a MariaDB server.

To give an idea of the size of OrderFlow, it's main database has (at the last count) **167** tables, plus another **76** views.

Tables for data entities are created with auto-incrementing identifiers, to allow the application to unambiguously identify entity records. Some of these entities also have **external reference** fields, which hold a reference to the entity that may be used outside of OrderFlow. For example, an *order*'s external reference would be known by (indeed *supplied* by) a shopping cart system.

Relationships are maintained by foreign key associations (for simple 'many-to-one' relationships), or join tables (for the more complex 'many-to-many' relationships). This ensures that the application's data cannot lose its integrity.

Some entity records are never deleted from the system, since they hold some referential integrity for other entity records. For example, a *location* is not deleted if there have been stock changes that refer to that location. Instead, if the user wishes to delete such an entity, its *deleted* flag is set, and it is never subsequently used by the application.

Indexes are applied to certain columns in specific tables. This speeds up data access for the specific queries that the system queries performs.

7.1. Traceability

Changes to entity tables are recorded with varying levels of traceability. Some changes just record the time at which the change was made, whereas others additionally record which user made that change. This can be useful when determining when and who made changes to both transactional and configuration entities on the system.

7.2. Concurrency

To avoid a near-concurrent update of an entity record overwriting an earlier update (by a different user), entity records can be *versioned*. This allows the application to adopt an optimistic locking strategy with regard to data updates - an entity record's version is checked before each update and incremented if it differs from the expected value.

8. Non-functional Aspects

This section deals with some of the non-functional elements of the OrderFlow architecture, in particular in areas of *monitoring*, *performance* and *scalability*.

8.1. Monitoring

OrderFlow's performance and health status can be closely **monitored** from the desktop *performance dashboard*, which shows requests that take a long time to serve, and other performance metrics.

Statistics on the performance of the system are automatically recorded in fifteen minute segments. These statistics cover both the time taken to serve different HTTP requests over the web as well as the time taken to call data layer methods.

In addition, key performance indicators such as CPU and memory utilisation are monitored in all of our hosted environments, to allow for early warning of performance related issues.

8.2. Performance

At Realtime Despatch we are very mindful of the importance of application performance in ensuring a rapid application response times and a good experience for OrderFlow users. We address this issue by both developing features to target application performance, and tuning environments to make the best use of the available hardware and software.

8.2.1. Application Features

OrderFlow includes a number of features which help the application to scale well even in hardware constrained environments.

- a large amount of *cacheing*, used to reduce the amount of unnecessary work that the system needs to do to carry out its functions. Cacheing on the system is used in many places, from data layer entities right through to the output of dashboard reports that appear on the user interface.
- built-in limits to the amount of data that can be returned through searches and reports.
- fine-tuned use of database indexes to ensure that queries run as efficiently as possible.
- advanced *data archiving* features, allowing for ageing data to be exported and removed from the system in an orderly fashion, helping to ensure that data volumes do not grow unnecessarily.
- an advanced scheduling capability that allows most long running operations to run in the background, in some cases at times when the application is less heavily used.

8.2.2. Tuning Strategies

The Java Virtual Machine (JVM) in which OrderFlow runs can be tuned to provide the right amount of heap and non-heap memory for the needs of the environment.

Tuning it also possible at the database level, particularly to ensure that the available system memory is used effectively. Appropriate use of memory settings at the database level is very helpful in ensuring that the database engine can make the most effective use of cacheing, making sure that data access is fast.

8.3. Scalability

The strategies described in the previous section allow a very simple hardware configuration to be used to support all but the largest environments. This allows the application to scale *vertically* to accommodate increasing performance requirements.

Beyond a certain point, some *horizontal* scaling is necessary. The exact strategies that will be employed in a particular environment will depend on the bottlenecks that need to be addressed, and may involve actions at the application as well as database layer.

The main strategy used on the application tier is *clustering*. On the data tier, various strategies can be employed.

8.3.1. Clustering

OrderFlow can be configured to run in a **clustered** configuration, in which multiple instances of the OrderFlow application are run in a single *load balanced* environment. This allows the load on the application server to be spread over multiple JVM processes or even physical servers.

Communication between the nodes, where required, is achieved using the Hazelcast clustering technology. This allows, for example:

- for caches to be kept consistent across the cluster
- for process level locking to occur across all nodes in the cluster rather than just a single node

Note that application tier clustering requires the use of a load balancer configured to support 'sticky sessions', with each request within a user session directed to the same JVM throughout the lifetime of that session.

8.3.2. Data Scaling

A number of strategies can be employed to scale the data tier in an OrderFlow environment.

- the main database and the blob database can be split into separate physical environments.
- database *replication* can be used with a master/slave configuration, in which all non-transactional requests (such as for reporting data) can be routed to one of multiple read-only replicas.

An option that has yet to be explored is the use of a *clustered* database server, which offers horizontal scalability within the database server itself. This could potentially address the need for extreme cases where write operations to the database need to be clustered across multiple physical servers.

8.4. Testing

Although not strictly an architectural aspect, it is worth mentioning the approach to **testing** OrderFlow's functionality.

From the bottom up, testing is crucial delivering a quality system that is robust and performant. OrderFlow's developers include three main layers of testing when delivering functionality, which are expanded upon in the following sections.

All tests are automated, and are run regularly to ensure that OrderFlow remains fully-functional and 'ready to build' at short notice.

8.4.1. Unit Testing

Adopting a Test-Driven approach to development, most logical code is covered by Unit Tests, using the JUnit testing framework. This enforces thorough consideration of all aspects of new features, and protects against errors being introduced by changes to existing features. Unit tests not only form a comprehensive regression test suite, but also serve to directly document the code in the form of readable tests.

8.4.2. Integration Testing

Since unit tests cannot efficiently cover interaction *between* different classes, the unit tests are complemented by a suite of Integration Tests, which cover this interaction more effectively and realistically.

Using the Impala dynamic modular productivity framework, sections of OrderFlow can be loaded in order to test co-operative areas of functionality. This provides a high degree of confidence in the behaviour of the real system, as it runs the *actual* code that is deployed in a production system.

Integration tests not only give functional assurance; they also prove that the system is *structured* correctly, and provide robust regression checks to ensure that no changes have had inadvertent side-effects in unexpected parts of the system.

8.4.3. Graphical User Interface Testing

To give an extra layer of confidence in the system's behaviour, and to cover parts of the user interface layer that may not have been comprehensively covered, OrderFlow has a suite of automated Graphical User Interface (GUI) tests. These exercise OrderFlow's user interfaces in the way that end users are expected to, and so again provide a level of functional and regression testing that unit and integration tests cannot provide.

We use Selenium to automate user interface testing from web browsers.